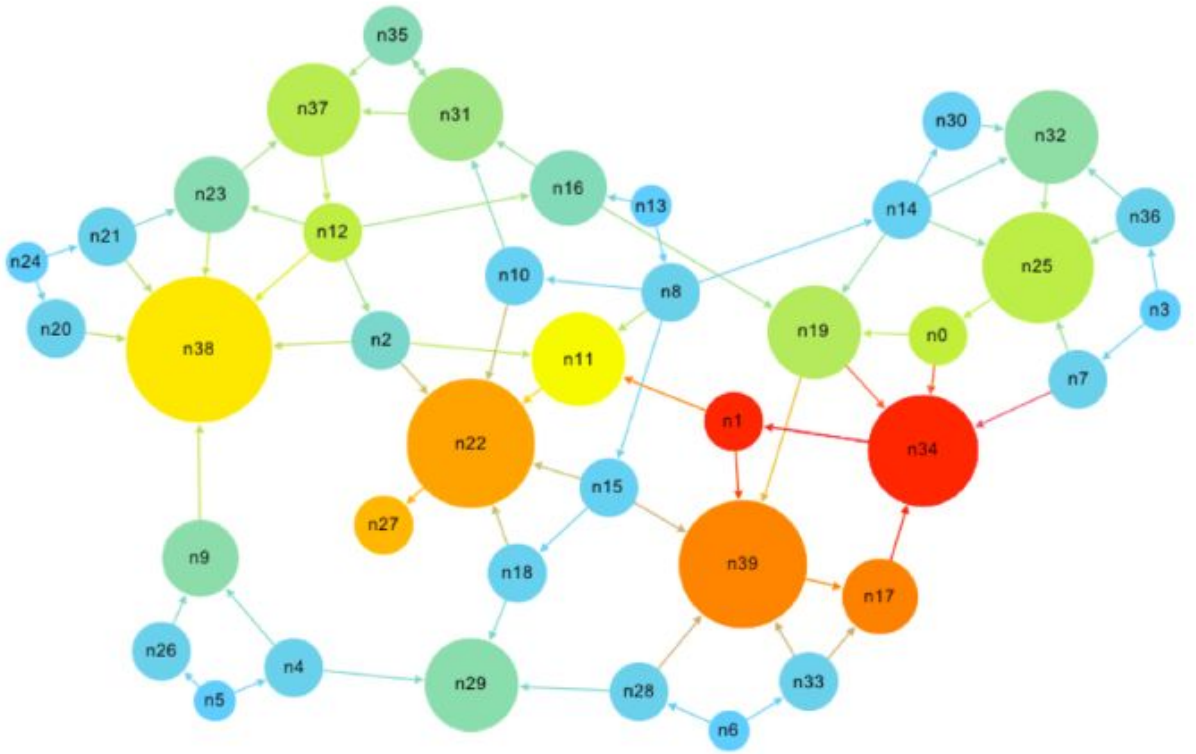


# Mapreduce-PageRank

*Introduction to Parallel and Distributed Programming*



04.05.2020

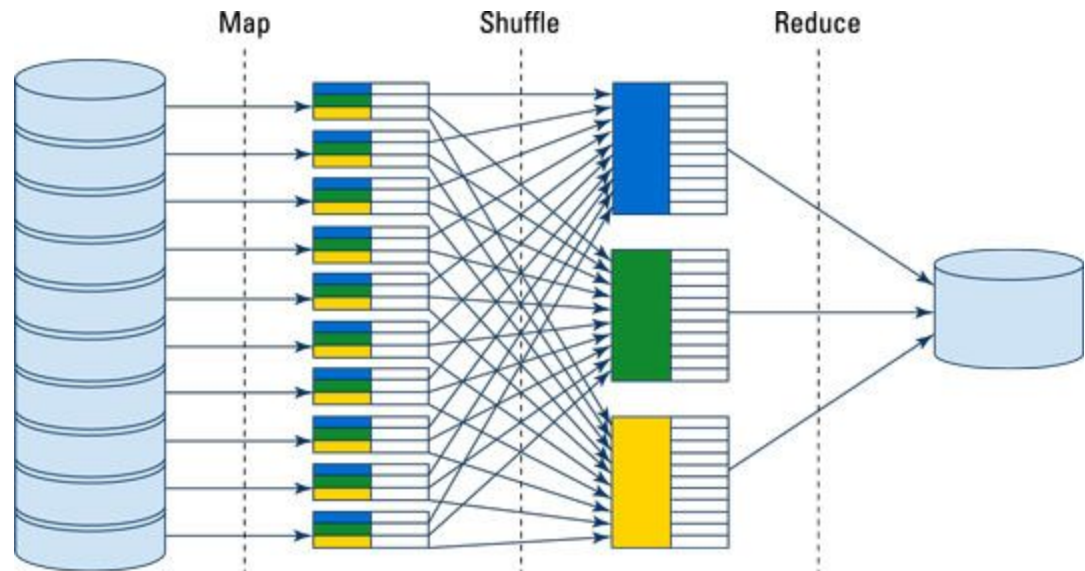
**Prateek Garg**

2017CS10360

## About MapReduce Paradigm

*MapReduce* is a programming paradigm that was designed to allow parallel distributed processing of large sets of data, converting them to sets of tuples, and then combining and reducing those tuples into smaller sets of tuples. In layman's terms, MapReduce was designed to take big data and use parallel distributed computing to turn big data into little- or regular-sized data. It works in following steps:

1. Map the data
  - a. The incoming data must first be delegated into key-value pairs and divided into fragments, which are then assigned to map tasks.
  - b. Each computing cluster — a group of nodes that are connected to each other and perform a shared computing task — is assigned a number of map tasks, which are subsequently distributed among its nodes.
2. Intermediate steps
  - a. Upon processing of the key-value pairs, intermediate key-value pairs are generated. The intermediate key-value pairs are sorted by their key values, and this list is divided into a new set of fragments.
  - b. Whatever count you have for these new fragments, it will be the same as the count of the reduce tasks
3. Reduce the data
  - a. Every reduce task has a fragment assigned to it. The reduce task simply processes the fragment and produces an output, which is also a key-value pair.
  - b. Reduce tasks are also distributed among the different nodes of the cluster. After the task is completed, the final output is written onto a file system.

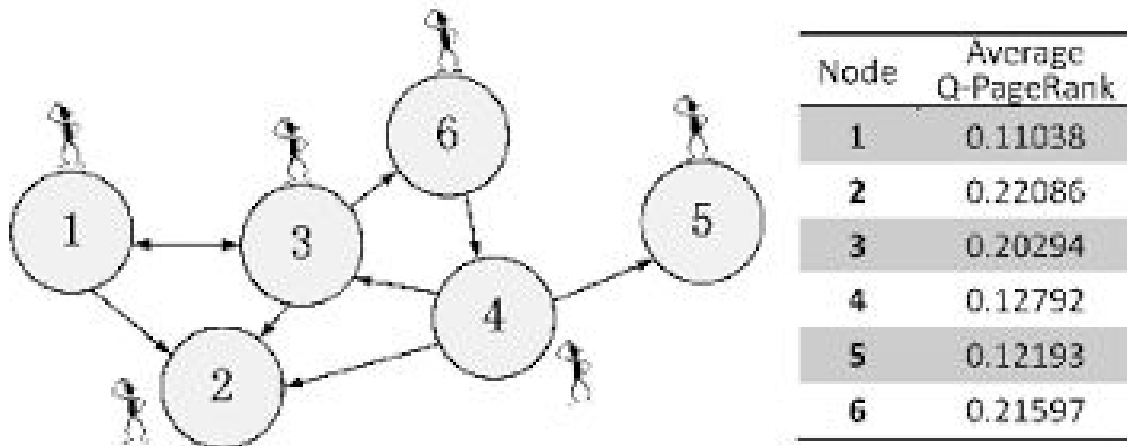


## About PageRank Algorithm

The PageRank algorithm outputs a probability distribution used to represent the likelihood that a person randomly clicking on links will arrive at any particular page. PageRank can be calculated for collections of documents of any size. It is assumed in several research papers that the distribution is evenly divided among all documents in the collection at the beginning of the computational process. The PageRank computations require several passes, called “iterations”, through the collection to adjust approximate PageRank values to more closely reflect the theoretical true value. The **basis for PR calculations** is the assumption that every website on the *World Wide Web* has certain importance which is indicated by the PageRank (0 being the least and 10 being the most important). The PageRank is calculated by the number and value of incoming links to a website.

1. Initially, one link from a site equaled one vote for the site that it was linked to. However, later versions of the PageRank set 0.25 as the initial value for a new website (based on an assumed probability distribution between 0 and 1).
2. The value of inbound links is determined by the amount of outbound links from the linking site and its PR. That is, because the PageRank of the linking site is divided by the total number of outbound links the page has.
3. A page that is linked to several relatively important websites (those with high

PageRank themselves) has value, while a website that has no incoming links is considered unimportant. The more the incoming links from high value websites you get, the better it is for your website.



## How can we implement PageRank using MapReduce ?

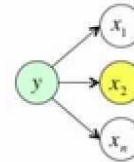
One of the most popular algorithms in processing internet data i.e. web pages is page rank algorithm which is intended to decide the importance of a webpage by assigning a weighting value based on any incoming link to that webpage. However, the large amount of internet data may lead to a computational burden in processing those page rank algorithms. Our algorithm can be decomposed into three processes, each of which is implemented in one Map and Reduce job.

1. We produce the page and its outgoing links as key and value pair, respectively, as well as total dangling node's weight and total amount of pages.
2. We next calculate the probability of each page and distribute this probability to each outgoing link evenly.
3. Each of the outgoing weights are shuffled and aggregated based on similarity of page title to update a new weighting value of each page.
4. In the calculation we consider the dangling node. In the end, all of the pages are descendingly sorted based on their weighting values. From the experimental result, we show that our implementation has output with reasonable ordering results.

# PageRank using MapReduce

- **Mapper:**  $\langle y, \{x_1 \dots x_n\} \rangle$  node + out-links

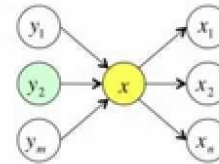
- for  $j=1 \dots n$ : **emit**  $\langle x_j, \frac{PR(y)}{out(y)} \rangle$
- **emit**  $\langle y, \{x_1 \dots x_n\} \rangle$



- **Reducer:**  $\langle x, \left\{ \frac{PR(y_1)}{out(y_1)}, \dots, \frac{PR(y_m)}{out(y_m)} \right\} \rangle$  node +  $\Delta PR$  from in-links

- **compute:**  $PR(x) = \frac{1 - \lambda}{N} + \lambda \sum_{y \rightarrow x} \frac{PR(y)}{out(y)}$

- for  $j=1 \dots n$ : **emit**  $\langle x_j, \frac{PR(x)}{out(x)} \rangle$



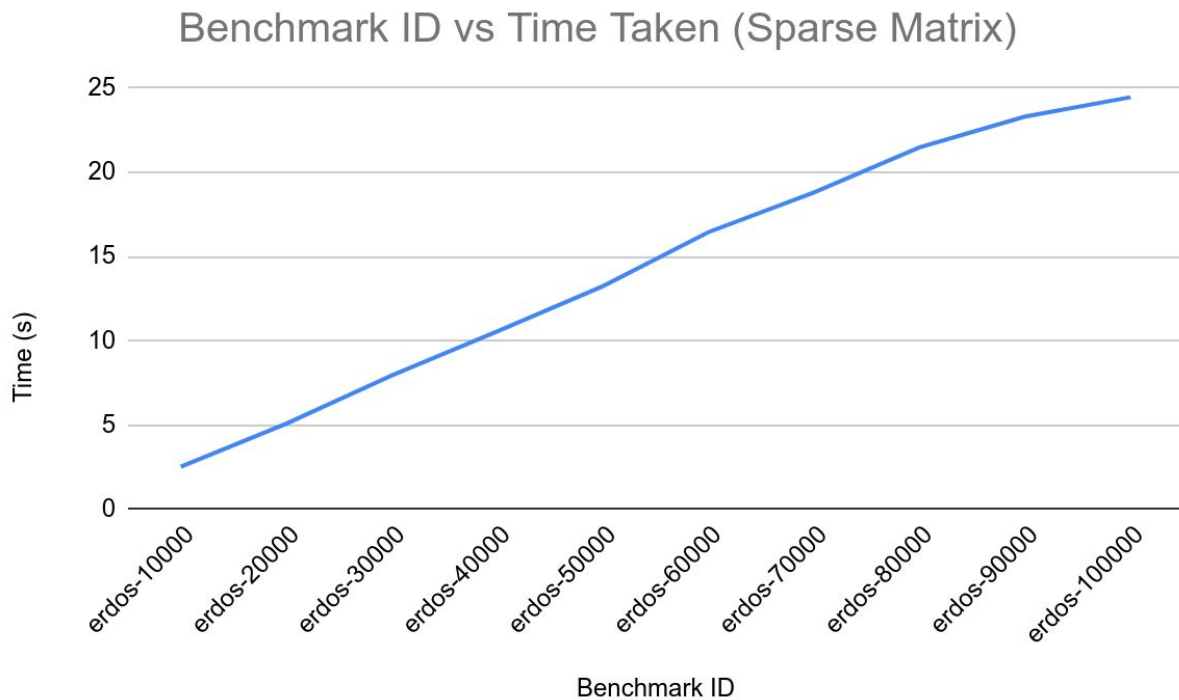
- Results go into another reducer (multi-step job)



Copyright © 2013, Victor Lavyenko

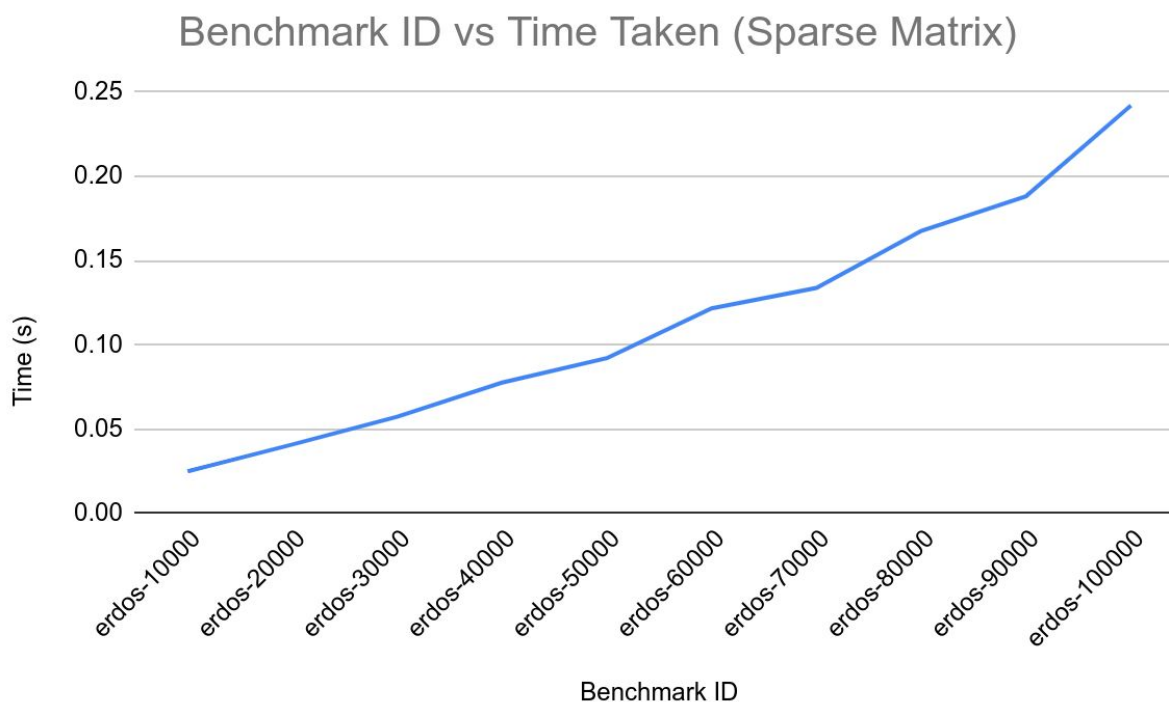
## Implementing MapReduce using C++ Library:

We used the [MapReduce API](#) for implementing our PageRank algorithm. Though we see a large difference in time, among this and the MPI implementations, the implementation returns correct results for large graphs.



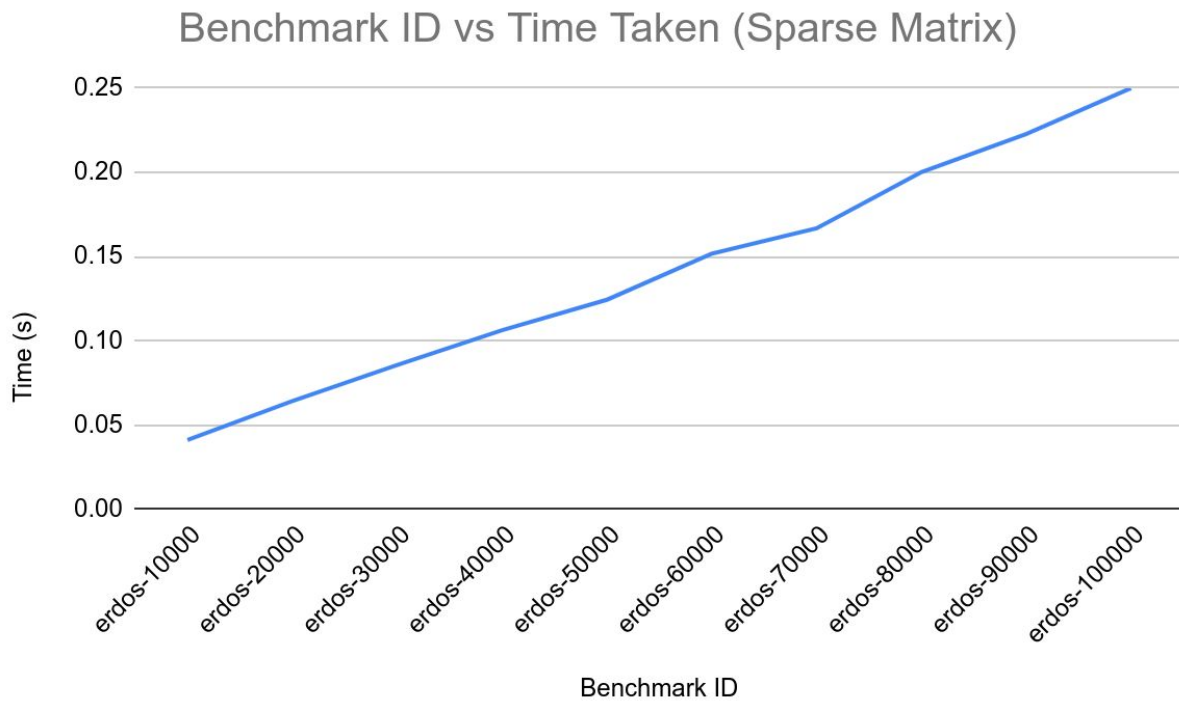
## Implementing MapReduce using our own MPI Library

We made our own MapReduce library for the specific pagerank task. The library performs better than the baseline [mrmmpi](#) library. We suspect this happening because our implementation does not use the intermediate key-conversion task in the MapReduce algorithm. Thus making it faster. We have also done a dangling\_nodes optimization and included it in the MapReduce algorithm which further makes it faster.



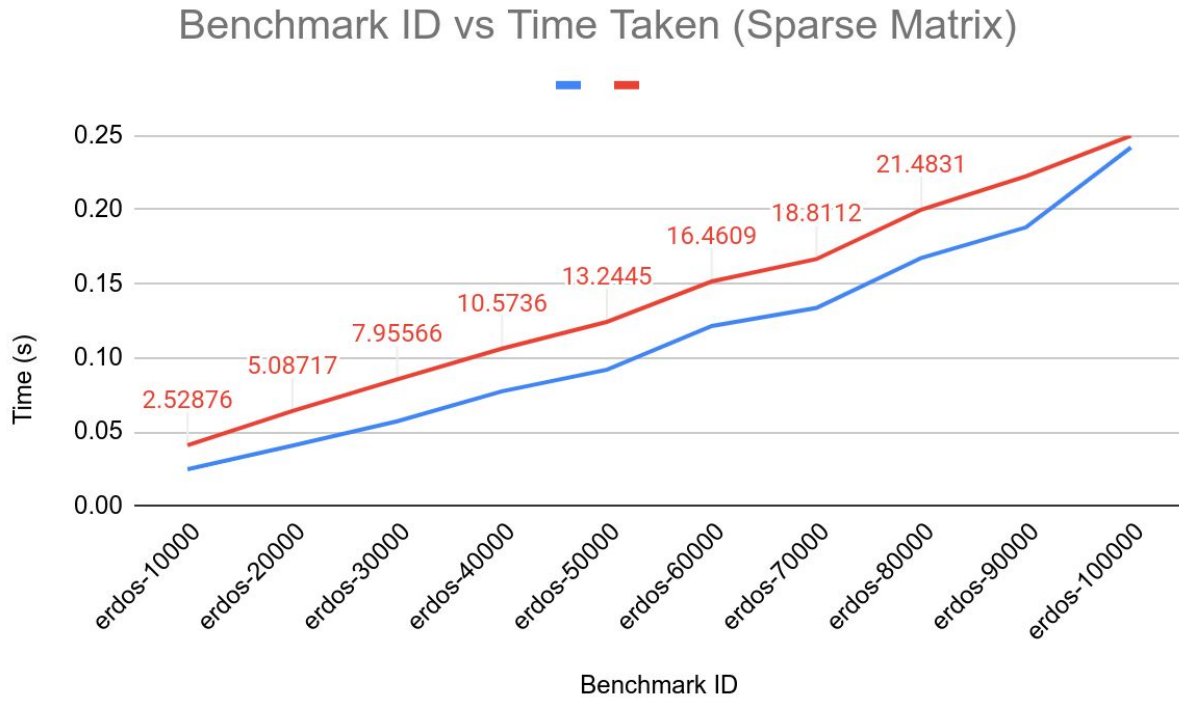
## Implementing MapReduce using existing MPI Library

The baseline implementation of pagerank algorithm, runs great on large graphs, though it uses an intermediate step in between Map and Reduce steps which makes it just a little less efficient than our implementation.





## Experimental Results for Latencies in each case:



BenchmarkID	mr-pr-cpp	mr-pr-mpi	mr-pr-mpi-base
erdos-10000	2.52876	0.024931	0.0412049
erdos-20000	5.08717	0.040853	0.064306
erdos-30000	7.95566	0.057307	0.085608
erdos-40000	10.5736	0.0775971	0.10635
erdos-50000	13.2445	0.092087	0.124385
erdos-60000	16.4609	0.121628	0.151715
erdos-70000	18.8112	0.133773	0.166766
erdos-80000	21.4831	0.167594	0.200131
erdos-90000	23.3091	0.188226	0.222713
erdos-100000	24.458	0.242069	0.249926

## Explanation for the observations:

The blue line shows our custom MPI MapReduce Algorithm and the Red curve shows the baseline. We obtained a better performance, attributing to, dangling\_nodes optimization, removal of intermediate step and copy of whole map structure in every iteration.

## Special Remarks (if any):

We ran all the experiments assuming a couple of distributed machines on x\_86 architecture. The machine specifications are as follows.

CPU op-mode(s): 32-bit, 64-bit  
Byte Order: Little Endian  
CPU(s): 8  
On-line CPU(s) list: 0-7  
Thread(s) per core: 2  
Core(s) per socket: 4  
Socket(s): 1  
NUMA node(s): 1  
Vendor ID: GenuineIntel  
CPU family: 6  
Model: 142  
Model name: Intel(R) Core(TM) i7-8550U CPU @ 1.80GHz  
Stepping: 10  
CPU MHz: 800.074  
CPU max MHz: 4000.0000  
CPU min MHz: 400.0000  
BogoMIPS: 3984.00  
L1d cache: 32K  
L1i cache: 32K  
L2 cache: 256K  
L3 cache: 8192K  
NUMA node0 CPU(s): 0-7